Chapter 15. A First-Person Shooter (FPS)

In chapter 14, we developed an application containing a stand-alone gun that shot laser beams. This chapter continues the peace-loving theme, so common in games programming, by putting a gun into the user's hand, and allowing him/her to do the shooting. This type of game, typified by Doom and Quake, is called a *first-person shooter* (FPS).

Figure 1 contains two screenshots for the FPShooter3D application: the first has the user strafing a robot – only laser beams that come close enough to the robot explode. The second screenshot shows a similar scene after the user has 'sneaked' behind and upclose to the robot.

A laser beam that misses the target vanishes after travelling a certain distance. To simplify the coding a little, no sounds are played. Also, the explosions do no 'harm' to the robot, who just keeps standing there.

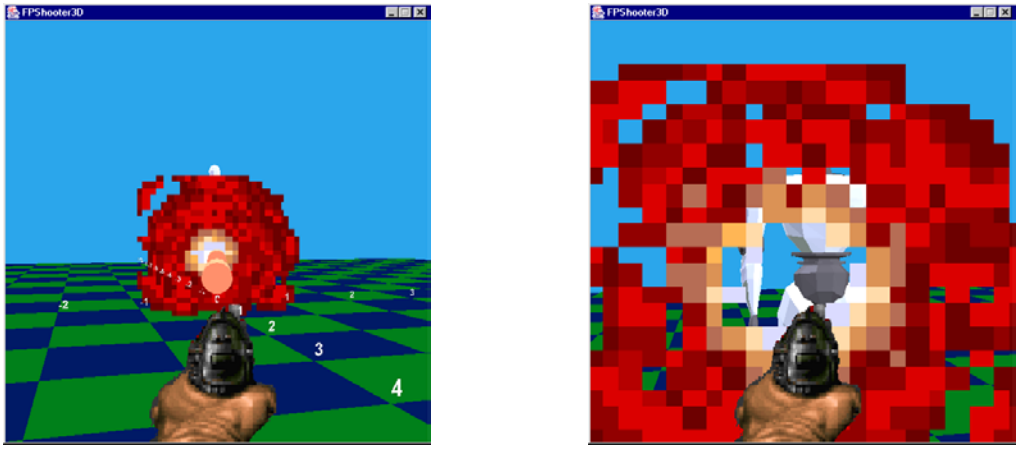

Figure 1. FPS Diplomacy.

Java 3D and Java features illustrated by this example:

- the "gun-in-hand" image is attached to the user's viewpoint, so travels with the user;
- keyboard navigation moves the user's viewpoint around the scene;
- multiple laser beams and explosions can be in the scene at the same time. (The example from chapter 14 only allows one laser beam on-screen at a time.)

This chapter reuses (or slightly changes) the techniques of chapter 14 for creating the laser beams (thin red cylinders) and the animated explosions, so it's advisable to read that chapter before this one.

1. UML Diagrams for FPShooter3D

Figure 2 gives the UML diagrams for all the classes in the FPShooter3D application. The class names and public methods are shown.

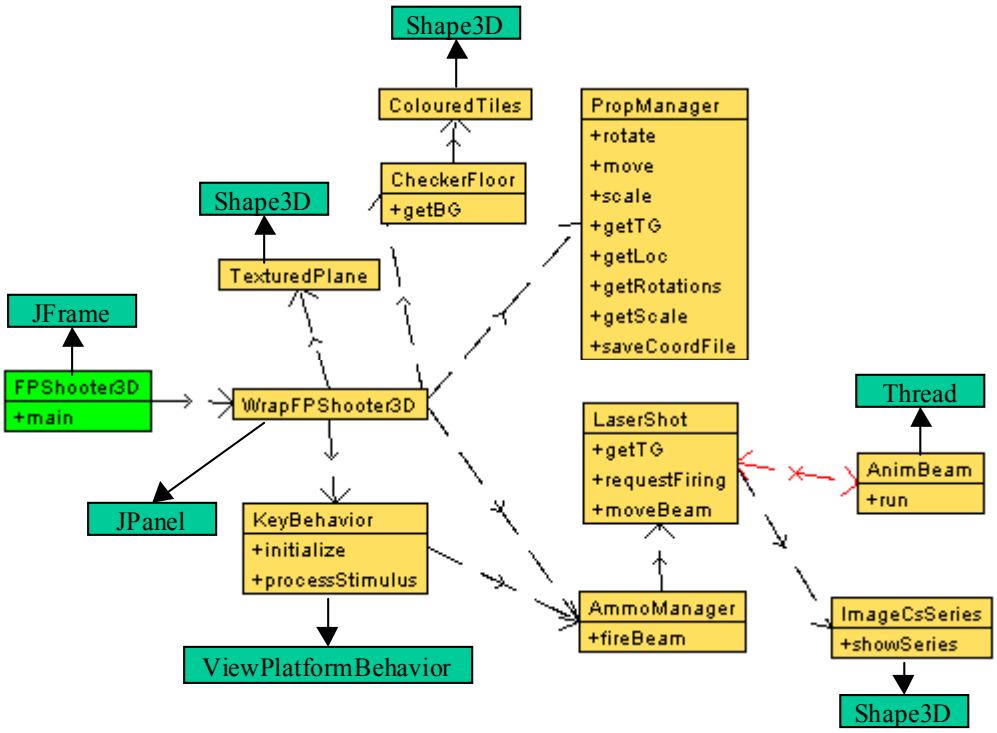

Figure 2. UML Class Diagrams for FPShooter3D.

FPShooter3D is the top-level JFrame for the application, and very similar to earlier JFrame classes.

WrapFPShooter3D creates the 3D world as usual, and adds the target and viewer elements.

The target (a robot from Coolrobo.3ds) is loaded with PropManager, a class first seen in chapter 9 in the Loader3D example.

The gun-in-hand image is a transparent GIF loaded and displayed by a TexturedPlane object.

CheckerFloor and ColouredTiles manage the checkboard, as in earlier examples. They were first encountered in chapter 8 in Checkers3D.

A LaserShot object represents a single beam and its (possible) explosion. The sequence of explosion images are stored in an ImageCsSeries object, a slight variant of the ImagesSeries class from chapter 14. The animation of the beam and subsequent explosion is initiated from an AnimBeam thread.

AmmoManager manages a collection of LaserShot objects, which allows the application to display several beams/explosions concurrently.

2. The Target

createSceneGraph() in WrapFPShooter3D loads the target using PropManager, places it in the scene, and records its location:

```
PropManager propMan = new PropManager(TARGET, true);
sceneBG.addChild( propMan.getTG() );
Vector3d targetVec = propMan.getLoc();
System.out.println("Location of target: " + targetVec );
```

`TARGET` is "Coolrobo.3ds", and the `true` argument to the PropManager constructor means that there is a "Coords" data file which fine-tunes its position. The robot appears facing along the positive z-axis, with its feet resting on the XZ plane, in the middle of the floor.

targetVec holds the center point of the shape, and is used later to calculate if a beam is close enough to the robot to trigger an explosion.

3. The View Branch Graph

The view branch graph is used for the viewer side of the scene; its main elements are shown in Figure 3.

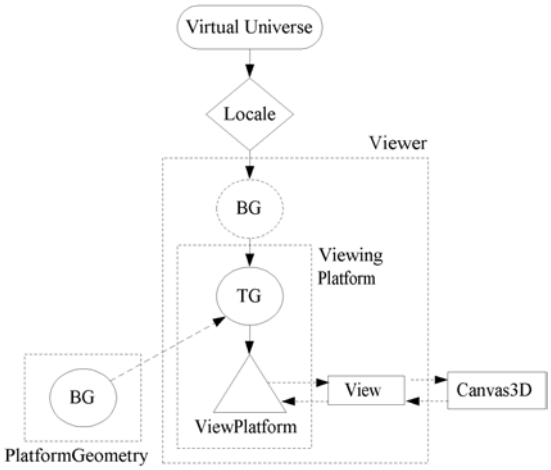

Figure 3. The View Branch Graph.

The ViewPlatform node is where the viewer is 'located' in the scene, and transforms applied to the TransformGroup (or groups) above the ViewPlatform will make the viewpoint move. It is possible to have multiple TransformGroups and BranchGroups in the path between the Locale and the ViewPlatform.

Geometry can be attached to the TransformGroups, which appear at the viewpoint and move along with the viewer. This functionality is useful for coding on-screen representations of the user (often called *avatars*).

If the SimpleUniverse utility class is employed to create the scene graph then a view branch is created (semi-)automatically. Also, SimpleUniverse supports 'simplified' access to the graph via utility classes: Viewer, ViewingPlatform, and PlatformGeometry, and others.

We have frequently employed ViewingPlatform in order to move the starting position of the viewpoint, and attached an OrbitBehavior object to convert mouse presses into viewpoint movements (see the end of chapter 8 for details). An example using OrbitBehavior:

```
OrbitBehavior orbit =
        new OrbitBehavior(c, OrbitBehavior.REVERSE_ALL);
orbit.setSchedulingBounds(bounds);

ViewingPlatform vp = su.getViewingPlatform();
vp.setViewPlatformBehavior(orbit);
```

The behaviour is attached to the ViewingPlatform with the setViewPlatformBehavior () method.

OrbitBehavior is a member of a useful set of classes for implementing behaviours that affect the TransformGroup above the ViewPlatform. The hierarchy is shown in Figure 4.

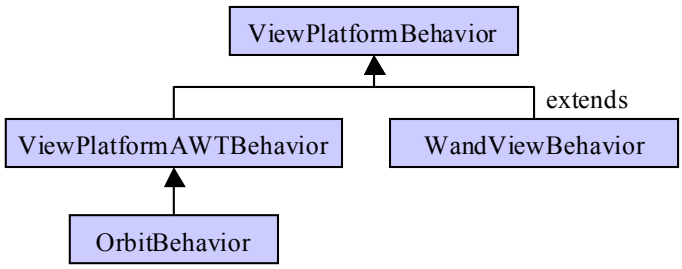

Figure 4. ViewPlatformBehaviour and subclasses.

ViewPlatformBehavior is an abstract class which supports basic functionality; it is a subclass of Behavior.

ViewPlatformAWTBehavior catches AWT events and places them in a queue. While there are pending events or mouse motion, the behavior will wake up every frame, and call processAWTEvents() and integrateTransforms().

WandViewBehavior manipulates the transform using a motion-tracked wand or mouse equipped with a six degree of freedom (6DOF) sensor.

The source code for all these classes can be found in the java3d-utils-src.jar file.

4. Initialising the Viewpoint

WrapShooter3D calls initUserControls() in order to configure the viewpoint. The method carries out four main tasks:

- sets up the user's gun-in-hand image;
- positions the user's initial viewpoint;
- calls AmmoManager to prepare beams and explosions;
- creates a KeyBehavior object to process keyboard input.

The initUserControls() method:

```
private void initUserControls(Vector3d targetVec)
{
  // add a 'gun in hand' image to the viewpoint
  ViewingPlatform vp = su.getViewingPlatform();
  PlatformGeometry pg = gunHand();
  vp.setPlatformGeometry(pg);

  // position starting viewpoint
  TransformGroup steerTG = vp.getViewPlatformTransform();
  Transform3D t3d = new Transform3D();
  steerTG.getTransform( t3d );
  t3d.setTranslation( new Vector3d(0, 1, Z_START) );
  steerTG.setTransform(t3d);

  // create ammo (beams and explosions)
  AmmoManager ammoMan =
          new AmmoManager(steerTG, sceneBG, targetVec);

  // set up keyboard controls
  KeyBehavior keyBeh = new KeyBehavior( ammoMan );
       // keyBeh can ask the ammoManager to fire a beam
  keyBeh.setSchedulingBounds(bounds);
  vp.setViewPlatformBehavior(keyBeh);
}  // end of initUserControls()
```

5. A Gun in the Hand is Worth …?

The call to gunHand() inside initUserControls() hides the creation of a TexturedPlane object just in front of, and below, the user's viewpoint:

```
private PlatformGeometry gunHand()
{
  PlatformGeometry pg = new PlatformGeometry();
  // define a square of sides 0.2f, facing along the z-axis
  Point3f p1 = new Point3f(-0.1f, -0.3f, -0.7f);
  Point3f p2 = new Point3f(0.1f, -0.3f, -0.7f);
  Point3f p3 = new Point3f(0.1f, -0.1f, -0.7f);
  Point3f p4 = new Point3f(-0.1f, -0.1f, -0.7f);
  TexturedPlane tp = new TexturedPlane(p1, p2, p3, p4, GUN_PIC);
  pg.addChild( tp );
  return pg;
}
```

PlatformGeometry is nothing more than a detachable BranchGroup (easily confirmed by looking at its source code in java3d-utils-src.jar). It is added to the ViewPlatform's TransformGroup by a call to ViewingPlatform's setPlatformGeometry().

TexturedPlane is a Shape3D holding a four point QuadArray; the constructor is called with the points, and with a transparent GIF that will be pasted onto the quad's front face. TexturedPlane can be viewed as a simplification of the ImagesSeries class of the last chapter, which lays a sequence of GIFs over a quad to exhibit an animation.

The most complicated aspect of using TexturedPlane is determining suitable coordinates, which is mostly a matter of trial-and-error until the image appears at the

bottom edge of the screen. It is helpful to remember that the (x,y) coordinate pair (0,0) corresponds to the middle of the canvas, and that negative z-coordinates are further *into* the scene. For example, with the following points:

```
Point3f p1 = new Point3f(-0.1f, -0.1f, -1.7f);
Point3f p2 = new Point3f(0.1f, -0.1f, -1.7f);
Point3f p3 = new Point3f(0.1f, 0.1f, -1.7f);
Point3f p4 = new Point3f(-0.1f, 0.1f, -1.7f);
```

The gun-in-hand image is located as shown in Figure 5: in the center of the screen, and further into the scene.

Figure 5. Gun-in-hand at a new Initial Location.

An alternative approach is to use Viewer and ViewerAvatar. A fragment of code illustrates the approach:

```
TexturedPlane tp = new TexturedPlane(p1, p2, p3, p4, GUN_PIC);
ViewerAvatar va = new ViewerAvatar();
va.addChild( tp );
Viewer viewer = su.getViewer();
viewer.setAvatar(va);
```

ViewerAvatar plays the same role as PlatformGeometry: it is subclass of BranchGroup, and its object can be detached from the scene at run time. setAvatar() connects the BranchGroup to the transform above the ViewPlatform node.

Since any shape can be attached to the viewpoint, why did I choose a QuadArray acting as a surface for a transparent GIF? One reason is efficiency – it would require a complex shape to represent a hand and a gun, together with suitable colouring/textures. In contrast, the GIF is only 5K in size.

Another advantage is that occlusion is less likely to happen – occlusion occurs when the image at the viewpoint intersects with a shape in the scene and is partially hidden. Since the GIF is flat, the viewpoint must move right up to a shape before occlusion happens.

3D special-effects, such as the gun recoiling, can still be coded, by using multiple gun-in-hand images, and employing a variant of the ImagesSeries class to animate them.

Section 6.3 of "Java 3D Programming" by Daniel Selman has two avatar examples (PlatformTest.java and AvatarTest.java) using the techniques described here. In PlatformTest.java, the avatars are cones with text labels, and in AvatarTest.java the avatar is a large cube.

6. AmmoManager

A drawback of the shooting application in chapter 14 is that only a single laser beam can appear at a time, and if another explosion is triggered before the previous one has finished then the animation (and sound effects) may be disrupted.

AmmoManager fixes these problems by creating a collection of beams and explosions; several beams/explosions can now appear at the same time because they are different objects.

Each beam and explosion is represented by a LaserShot object, and AmmoManager's constructor creates `NUMBEAMS` of them (20), and adds them to the scene.

```
public AmmoManager(TransformGroup steerTG,
                BranchGroup sceneBG, Vector3d targetVec)
{
  // load the explosion images
  ImageComponent2D[] exploIms = loadImages("explo/explo", 6);

  shots = new LaserShot[NUMBEAMS];
  for(int i=0; i < NUMBEAMS; i++) {
    shots[i] = new LaserShot(steerTG, exploIms, targetVec);
    // a LaserShot represents a single beam and explosion
    sceneBG.addChild( shots[i].getTG() );
  }
 }
```

An explosion animation uses six GIFs, and it would be very inefficient if each LaserShot object loaded those images. Instead, AmmoManager loads them once, with loadImages(), and passes an ImageComponent2D array to each LaserShot object.

Shooting the Gun

A beam is fired from the gun when the user presses 'f' on the keyboard. The KeyBehavior object captures the key press and calls fireBeam() in AmmoManager.

AmmoManager's managerial role is to hide the fact that there are `NUMBEAMS` LaserShot objects, and instead offer a single fireBeam() method, which fires a beam *if* one is available.

```
public void fireBeam()
{ for(int i=0; i < NUMBEAMS; i++) {
    if( shots[i].requestFiring() )
      return;
  }
}
```


The requestFiring() method returns true if the beam was free and has now been set in motion. It returns false if the beam was already in use before the request (i.e. already travelling or perhaps exploding).

This coding approach means that it is possible for all the LaserShot objects to be busy when the user types 'f', and so for nothing to happen. In practice however, this situation is unlikely to occur because a beam is busy for no more than 3-4 seconds, and there are 20 LaserShot objects available. From a gaming point of view, the possibility of running out of beams, albeit temporarily, may actually make the game play more interesting!

7. LaserShot

Each LaserShot object creates and manipulates a subgraph holding a beam (a thin red cylinder) and explosion (an animation laid over a quad). The subgraph shown in Figure 6 is created by makeBeam().

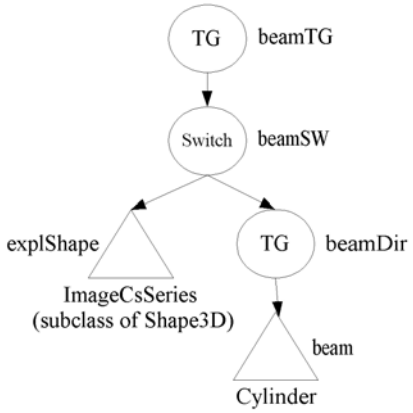

Figure 6. Scene Graph Branch for LaserShot.

The top-level TransformGroup, beamTG, is employed to move the beam (and explosion), and the Switch allows the beam, explosion, or nothing to be displayed. The beamDir TransformGroup is used to initially position and orientate the cylinder so it is pointing into the scene and appears close to the nozzle of the gun image.

The cylinder has a radius of 0.05 units, height 0.5; it is rotated by 90 degrees, moved down by 0.3, and into the scene by 0.25 units. This means that the center of its tail is located at (0,-0.3,0) relative to the coordinate space of beamTG.

The ImagesCsSeries class is virtually identical to the ImagesSeries class of chapter 14. The two differences are: the object does not load its own GIFs, instead an array of ImageComponent2D objects is passed to it in the constructor, and the shape's position is specified with a center point. The call to ImageScSeries' constructor is carried out in makeBeam():

```
// create explosion, centered at (0,0,0), size 2.0f
explShape = new ImageCsSeries( new Point3f(), 2.0f, exploIms);
```

The explosion is a child of the same TransformGroup as the beam (beamTG), and so placing its center at (0,0,0) means that it will appear at roughly the same place in beamTG's coordinate space as the tail of the beam.

The positioning of the beam is something of an 'art' since it depends on creating the illusion that it is coming from the gun, which in turn depends on the way that the gun-in-hand is drawn and placed on screen. Similarly, the positioning of the explosion is governed by where the center of the explosion is drawn in the GIF, and by where that center should be relative to the beam at explosion time.

Firing a Beam

AmmoManager calls LaserShot's requestFiring() method to utilise the beam. The request will only be accepted if the beam is not already in use, which is recorded by setting the inUse boolean.

If the LaserShot is not in use (inUse == false) then LaserShot starts an AnimBeam thread, and inUse is set to true.

```
public boolean requestFiring()
{ if (inUse)
    return false;
  else {
    inUse = true;
    new AnimBeam(this).start(); // calls moveBeam() inside a thread
    return true;
  }
}
```

The AnimBeam thread is extremely simple: it's sole purpose is to call the moveBeam () method back in the LaserShot object. By being called in a thread, the method will be executed without causing the rest of the application (e.g. KeyBehavior, AmmoManager) to wait.

moveBeam() incrementally moves the beam (and explosion) forward, starting from the current viewer position (steerTG). If the beam gets close to the target then the explosion is shown, otherwise the beam disappears after reaching a certain MAX_RANGE distance from the gun.

inUse is set to false again at the end of the method, allowing the LaserShot object to be again used by AmmoManager.

```
public void moveBeam()
{
  // position the beam at the current viewer position
  steerTG.getTransform( tempT3d );
  beamTG.setTransform( tempT3d);
  showBeam(true);

  double currDist = 0.0;
  boolean hitTarget = closeToTarget();
  while ((currDist < MAX_RANGE) && (!hitTarget)) {
    doMove(INCR_VEC);
    hitTarget = closeToTarget();
    currDist += STEP;
    try {
      Thread.sleep(SLEEP_TIME);
    }
    catch (Exception ex) {}
  }
```



```
    showBeam(false);     // make beam invisible
    if (hitTarget)
      showExplosion();   // if a hit, show explosion
    inUse = false;       // shot is finished
  }
```

INCR_VEC is the vector (0, 0, -1). It is repeatedly applied to the beam's TransformGroup, beamTG, by doMove() to move the beam away from the viewer's position. This works because the top-level transform for the beam (and explosion), beamTG, was set equal to steerTG before the loop began, which gives it the same starting position and orientation as the viewer.

The code for doMove():

```
  private void doMove(Vector3d mv)
  { beamTG.getTransform( tempT3d );
    toMove.setTranslation( mv );
    tempT3d.mul(toMove);
    beamTG.setTransform( tempT3d );
  }
```

The doMove and tempT3d references are global, to avoid the creation of temporary objects.

closeToTarget() does a simple comparison between the current position of the beam and the position of the target. This is complicated by the fact that the beam's location in the scene is affected by beamTG *and* by its initial transformation with beamDir.

A general-purpose solution is to use getLocalToVworld() on the beam shape to retrieve the overall transformation in terms of scene coordinates. This requires a capability bit to be set when the shape is created:

```
    beam.setCapability(Node.ALLOW_LOCAL_TO_VWORLD_READ);
```

The closeToTarget() method:

```
  private boolean closeToTarget()
  /* The beam is close if its current position (currVec)
     is a short distance from the target position (targetVec).
  */
  { beam.getLocalToVworld(localT3d);  // beam's trans in world coords
    localT3d.get(currVec);            // get (x,y,z) component

    currVec.sub(targetVec);    // calc distance between two positions
    double sqLen = currVec.lengthSquared();
    if (sqLen < HIT_RANGE*HIT_RANGE)
      return true;
    return false;
  }
```

The code is basically carrying out a bounding sphere test to see if the beam is within HIT_RANGE units of the center of the target. This is not particularly accurate, especially since the robot target is tall and thin rather than spherical.

8. Moving the Viewpoint

FPShooter3D moves the viewpoint by attaching a KeyBehavior object to the ViewPlatform's TransformGroup.

```
ViewingPlatform vp = su.getViewingPlatform();
        :
KeyBehavior keyBeh = new KeyBehavior( ammoMan );
     // keyBeh can ask the ammoManager to fire a beam
keyBeh.setSchedulingBounds(bounds);
vp.setViewPlatformBehavior(keyBeh);
```

KeyBehavior's internals are similar to earlier keyboard-based behaviours developed for the Tour3D and AnimTour3D examples of chapters 10 and 11.

One difference is that this class extends ViewPlatformBehavior so that it can work upon the ViewPlatform's TransformGroup, called targetTG. targetTG is available to the methods defined in KeyBehavior through inheritance.

Another difference is that the Tour3D and AnimTour3D behaviours pass the responsibility for movement and rotation to objects in the scene (i.e. to the sprites). KeyBehavior carries out those operations for itself, by manipulating targetTG.

KeyBehavior detects key presses of the arrow keys, optionally combined with the <alt> key, in order to move forward, back, left, right, up, down, and rotate around the y-axis. The 'f' key is used to request the firing of a beam.

processStimulus() calls processKeyEvent() when a key is detected:

```
private void processKeyEvent(KeyEvent eventKey)
{ int keyCode = eventKey.getKeyCode();
  if(eventKey.isAltDown())   // key + <alt>
    altMove(keyCode);
  else
    standardMove(keyCode);
}
```

altMove() and standardMove() are multi-way branches calling doMove() or rotateY() (or AmmoManager's fireBeam(), when the key is 'f'). rotateY() applies a rotation to targetTG:

```
private void rotateY(double radians)
{ targetTG.getTransform(t3d);
  toRot.rotY(radians);
  t3d.mul(toRot);
  targetTG.setTransform(t3d);
}
```

t3d and toRot are globals to avoid the overhead of temporary object creation.

Java 3D offers a KeyNavigatorBehavior utility class, designed to operate on the ViewPlatform transform. It responds to arrow keys, +, -, page up, page down, utilises

key presses and releases, and the elapsed time between them. KeyNavigatorBehavior makes use of KeyNavigator, another utility. The source code for both are in java3d-utils-src.jar file.

A typical invocation of KeyNavigatorBehavior:

```
TransformGroup vpTrans =
    su.getViewingPlatform().getViewPlatformTransform();
KeyNavigatorBehavior keybehavior =
    new KeyNavigatorBehavior(vpTrans);
keybehavior.setSchedulingBounds(bounds);
scene.addChild (keybehavior);
```

KeyNavigatorBehavior extends Behavior, and so requires the ViewPlatform TransformGroup to be explicitly passed to it, and can be added to any node in the scene graph.

KeyNavigatorBehavior is employed in the IntersectTest.java example in the PickTest/ directory of the Java 3D demos. Daniel Selman also uses it in his PlatformTest.java example in section 6.3 of "Java 3D Programming".

